Introduction to objects

History and motivation

Internally computers execute instruction one after another, and manipulate raw bits of data in memory. The addition of names, functions, variables etc in programming language is meant to *help programmers to do their job better*, not the computer.

When the first computers were built, the people in charge of designing programming languages started with very simple features: variables, arrays, procedures. All the entities named in a program existed side by side, so *programmers had to choose a different name for every new entity*. This early requirement that all names must be different wasn't a problem initially, when there were only few entities in a computer. However as software engineering matured, software became more and more complex and it became difficult for human programmers to keep track of the names of everything.

In order to create a more productive structure for programmers, computer scientists in the 1970's have invented a new way to think about programs.

In their view, programmers should imagine that they are writing code for a large collection of many micro-computers, where each micro-computer would only perform a very simple task with very few variables. This way, it was thought, the human programmer could do a better job, because on each of these micro-computers the programmer could focus on performing only one task, and doing it well.

Also, because each of these micro-computers would have its own procedures and variables, the programmer did not need to invent new names all the time: *the same name could designate different things* on different micro-computers. Using this abstraction, a large software application can be built by connecting multiple such micro-computers together in a network and sending messages between them.

For example, this was very powerful to write programs that simulate traffic on the road: one would program a micro-computer to simulate the behavior of one car, then another for another car, and so on, then one micro-computers per road segment, and each could be programmed separately. In two different cars, the same variable name "speed" could be reused to designate each car's speed. The messages between a "car" micro-computer and a "street" micro-computer would be "the car has entered this streed" or "the car has left this street". With this simple composition, the entire program could then imitate a complex traffic situation, and each individual part could be very simply implemented without requiring the programmer to know about the complex whole.

As it so happens, this idea was very successful. Programming in this way did indeed increase the productivity of programmers and enable more complex applications that worked with fewer errors.

The micro-computers in this story are what are now called *objects*: a *small unit of behavior* in a large computer system with its own variables and procedures, and *where names can have a local meaning which differs across different objects.*

Object-oriented languages

A programming language is said to be "object-oriented" when it promotes thinking in terms of objects when writing code. A further distinction is made between:

- languages that *support object orientation as an opt-in feature*, where programmers can ignore object orientation when starting to write code, and only use objects in some cases; this is the case for most programming languages, including C, C++, Python, PHP, OCaml, Matlab, R;
- languages that are *exclusively object-oriented*, where a programmer can only write code in terms of objects; this is the case with e.g. Smalltalk and Java.

In general, *an introductory course on programming should not use a language that is exclusively object-oriented* because it forces a teacher to talk about objects before talking about more important topics, like task decomposition, typing, etc. However, perhaps unfortunately, your education program requires us to start with Java, so you need to start caring about objects right now, next to the other topics which are independent from objects.

Starting with Java's objects

Because programmers usually write complex programs that do not use only one object, but rather many of them side by side, it is more practical to express in program code *how an entire family of objects should behave*, instead of writing the same code over and over for every object.

This generalization is so important that all object-oriented languages are designed with this in mind. So the primary construct in the language is not the object, but rather *the set of all possible objects that belong to the same family*. This set is then described in code as a *class*.

In other words, a *class is a type which describes a set of all possible objects that share a common behavior*. For example in the traffic simulation mentioned above, "car" is a class, which describes the behavior of every possible car in a general way.

We will see more of objects and classes in a later lecture; for now you can simply remember that a class is defined in Java using the following construct:

Class definition:

Syntax:

```
class <identifier> {
        [
            [ <variable/attribute declaration> ]*
            [ <function/method definition> ]*
        ]*
```

```
}
```

(The keyword "`class`", followed by an identifier, followed by an opening brace, followed by zero or more variable and/or method definition(s), followed by a closing brace)

Semantics: this construct defines a class named by the identifier at the top, so that all objects created in this class have their own copy of the variables but share the same method code.

For example:

```
class Car {
    double mass;
    double speed;

    void accelerate(double force, double timeDelta)
    { ... }
}
```

The variables defined inside a class are called *attributes*. The procedures or functions defined inside a class are called *methods*. The previous `Car` example has two attributes `mass` and `speed` and one method `accelerate`.

Attributes are "inside" every object created from the class: two different objects will have two different sets of variables.

The "methods" are connected to the idea of exchanging messages between objects as explained above: an object can "send" a message to another, and this will cause the execution of the corresponding method on the target object. The general construct for this is defined as follows:

Method invocation:

Syntax:

<expression> . <identifier> ([<expression> [, <expression>]*]?)

(An expression, followed by a dot ., followed by a name, followed by an opening parenthesis, followed by zero or more expressions separated by commas, followed by a closing parenthesis)

Semantics: evaluate all the expressions, then send a message to the object identifed by the first expression, that causes the procedure/function named by the name in 2nd position, inside the object class, to be called on the target object, with the remaining expressions as input arguments.

For example:

```
out.println("hello")
```

This evaluates first "`out`", which designates an object of class `PrintStream`; then sends a message to the object `out` which causes its method `println` to be called with one argument, the string "hello".

Program start-up and execution order

There is a clear advantage to programming languages that are not exclusively object-oriented: it is easy to understand where the program starts. It is quite simple: the first line in the program text is typically the first thing that runs when the execution begins.

This simplicity implies that a programmer can literally write the code in the same order it will be run, which makes the task of a beginner programmer quite comfortable.

Unfortunately this is not possible in languages that are exclusively object-oriented: the first line of a program there typically has nothing to do with program execution, but rather the definition of classes.

So how does Java, for example, know where the program starts?

The rule is quite simple: Java will look at all classes defined in the application (there may be more than one), and *searches for a class which has a special method called "main"*. If none is found, the program cannot start; if one is found, the execution starts there.

This method is "special" because it must be introduced in a very specific way, which is waaaay more complicated than most the "regular" functions/procedures you will ever use in your own programs:

```
class HelloWorld {

    public static void main(String[] args) {
        ...
    }
}
```

We will explain the "`public static void`" and the "`String[]`" later; for the time being, just remember that "main" must always have this specific form.

To summarize

Objects and classes will be presented in more details in a later lecture. For the time being, **just remember the following**:

- because you are using Java, your entire program must be structured using classes;
- a class begins with "`class` YourName `{`" and ends with "`}`", *this will be mandatory in all your Java code*;
- all variables and function/procedures in Java must be part of a class, written within the braces;
- at least one class in your program must have a "`main`" method, with a fixed form constrained by the language.

Important concepts

- programming languages exist to help programmers, not computers;
- choosing many different names is hard;

- *objects* define small unit of behavior in a large system;
- object orientation was successful because thinking in terms of objects decreases the naming problem and simplifies the definition of complex programs;
- the difference between object orientation as an opt-in feature (most languages) and exclusive object orientation (Java, Smalltalk, a few others);
- a *class* is a common definition for a family of objects;
- *attribute* = variable inside a class;
- *method* = function/procedure inside a class;
- in Java all program behavior must be part of a class;
- there must be a "main" method somewhere.

Further reading

- Introduction to Programming, sections 1.4-1.5 (pp. 8-13)
- Think Java, sections 15.1-15.2 (pp. 193-194)
- Absolute Java, section 1.1 (pp. 2-4)

Copyright and licensing